

From 1 to N DAQ Jobs

(for N not too big)

Brett Viren

February 12, 2021

Aside: Taxonomy

We need a way to speak precisely about some very similar things.

application an **executable file** on host disk

- (eg `daq_application`)

job an **application command line**

- (eg `daq_application --arg...`)

process a **running job**

- (eg seen in `ps -ef | grep daq_application`)

node a **process with an identity**

- (has been given *init* command)

We may bike shed labels but at least these four are distinct.

Where we are

- Single node with multiple modules.
 - ▶ JSON objects generated from `moo.otypes` classes in Python
 - ▶ Python classes themselves generated from `moo schema` in `Jsonnet`.
- Start single job by hand to make a process.
- Various means to “manually” send **command** to process.
- Each command has a **type** (`.id`) and a payload (`.data`) of type **any**
 - ▶ types: *init*, *conf*, *start*, *stop*, *scrap*, ...
 - ▶ structure of `.data` depends on value of `.id` and determined by schema
- The *init* command is **fixed structure** for all jobs
 - ▶ *init* “constructs” (process → node) in terms of **modules** and **queues** and soon **sockets**
 - ▶ Mostly consumed by `DAQModuleManager` but modules may also receive notice.
- The commands supports **command dispatch protocol**
 - ▶ Generally maps regex match on module **name** to module-level sub-object **payload**
 - ▶ Most commands use `" . * "` match and common or no payload
 - ▶ The *conf* command carries very module-specific payloads.
- `appfwk` command handling:
 - ▶ “command facility” (`stdin`, `file` and `rest`) accepts commands
 - ▶ `DAQModuleManager` command **payload** dispatched to modules
 - ▶ **no support for any kind of reply to a command**

Next, scale to multiple processes

My take-away of the essentials of Giovanna's proposal:

- Generate **per-job** *init* and *conf* command objects **as we do now**.
- **Collate** these across **N jobs** by their command type (*init*, etc)
 - ▶ Must retain the association of command → job
- Augment this collection with *boot* info (to enact job → process)
 - ▶ Then play *init* (process → node) and *conf*, *start*, etc

Prototypical CLI operations:

Run existing scripts to produce *init* and *conf* commands for each job.

- maybe also *start*, *stop*, etc, but these are simple enough to be produced later.

```
$ my_gen_proc1 > proc1.json
```

```
$ my_gen_proc2 > proc2.json
```

Collate into run-level configuration, details how coming up.

```
$ gen_run -r 42 proc[12].json > run-42.json
```

Apply run using a temporary/mock RC (exact nature still t.b.d.)

```
$ run-control run-42.json
```

How **NOT** to collate?

Invent some way to express `#include` in JSON

How **TO** collate?

Use Jsonnet `import`.

- Jsonnet may `import` JSON files statically (no computed filenames)
- `moO` has more flexible ways to get data of many formats in to Jsonnet via top-level arguments (“TLAs”) to a Jsonnet function.
- All info needed for collation is available in the per-job commands so we may write a Jsonnet program to perform any collation.
- Still need to supply *boot* info, run number, map from **job name** to its commands.

Example construction

I'll walk through one way to construct run config.

This exposes guts. We'd easily script it.

Consider it **just a prototype** for something more formal, likely implemented Python.

- 1 make **mock** per-job commands
- 2 collate commands and keeping **job name correlation**
- 3 build **run config** object

Example construction - mock per-job commands

FAKE, just for illustration replace full blow per-job command generator with:

```
local dummy(name) = [
  {id:"init",data:name+" init"},
  {id:"conf", data:name+" conf"}
];
function(procs) {
  [n+".json"]:dummy(n) for n in procs
}
```

Run like:

```
$ moo -A procs='["proc1","proc2"]' \
      compile -m . per-proc.jsonnet
$ cat proc1.json
```

```
[
  {
    "data": "proc1 init",
    "id": "init"
  },
  {
    "data": "proc1 conf",
    "id": "conf"
  }
]
```

Of course **real commands** have some mongo object hanging on .data!

Example construction - correlate commands and job name

Proposal calls for **collating by command id** (*init* etc).

We must first **correlate *job name** to each command object.

Let's take job name from file name with some jq hackery:

```
$ ls proc?.json
```

```
proc1.json  proc2.json
```

```
$ jq -n \
```

```
'reduce inputs as $s (.; .[input_filename|split(".")[0]] += $s)
  proc?.json  > run-procs.json
```

The `run-procs.json` now has all commands of all jobs and each command holds a `.job` attribute keeping the job name.

Example construction - collate and build *boot*

Use `moo` to provide data a `run.jsonnet` program to produce a final **run configuration object** with collated commands and *boot* part:

```
$ moo -A run=42 -A procs=run-procs.json run.jsonnet \  
> run-42.json
```

The `run-42.json` now contains:

- `.boot` minimal example of *boot* command
- `.inits` array of per-job *init* command
- `.confs` array of per-job *conf* command

Embellishments certainly still needed:

- exhaustively include also the `.starts`, `.stops`, etc commands.
- expand *boot* info beyond just example `.jobs` (names) and `.run`
- generate `rest://` URLs and `daq_application` command lines

run.jsonnet - just for reference

```
local boot(run, jobs) = {
    id: "boot", run: run, jobs: jobs
};

local select(cmdid, job, cmds) =
    [ c{job:job} for c in cmds if c.id == cmdid ];

local procs_to_cmds(procs, cmdid) =
    std.flattenArrays([select(cmdid, job, procs[job])
                       for job in std.objectFields(procs)]);

function(run, procs) {
    inits:procs_to_cmds(procs, "init"),
    confs:procs_to_cmds(procs, "conf"),
    boot:boot(run, std.objectFields(procs)),
}
```

Steps to make this useful

Step 1: Package the hackery into a short script to run as show above:

```
$ gen_run -r 42 proc[12].json > run-42.json
```

Step 2: there no step 2.

That's it except for adding needed embellishments mentioned above!

Summary

- We have more or less solid single-job DAQ config now.
 - ▶ True, users are still getting up to speed!
- Giovanna's proposal looks good for next-step scale-up!
 - ▶ But, let's use Jsonnet to aggregate instead of inventing some kind of JSON `#include` requiring some invented interpreter.
- A simple `jq` + `moO` tool-based construction demonstrated and seems sufficient for now.
- Let's gain experience with this,
 - ▶ understand the nature of the eventual minidaq jobs
 - ▶ extend it with **real** jobs as fodder
 - ▶ likely outgrow the `jq` + `moO` hackery and re-implement in Python
 - ▶ simultaneously lets us be flexible as we understand how the coming new CCM components (RC, PM, AC) will look.